Chapter 7: Introduction to database design

As a database designer, you want to create a design that faithfully models a real-world situation, can be updated accurately and efficiently, and is easy to modify. You must determine what information to store in the database and which groups of information belong together.

This section describes the principles of database design, emphasizing features and access methods specific to Unify DataServer/ELS. The principles of database design are outlined as follows:

1. Develop a preliminary list of the elements, or fields, you want to store in the database. To develop this preliminary list, you should begin by talking to the prospective users of the database to establish their needs.

   Database fields each have a name, a length, and a data type, such as STRING, NUMERIC, AMOUNT, or DATE. Field data types are described in Field Types in the Database Elements subsection and in Appendix A.

2. A relational database contains a collection of tables, each consisting of a group of related fields.

   Organize the preliminary list of fields into tables. Then subject the tables to a series of refinements. You refine tables by determining key fields and the dependencies among fields, and by eliminating unnecessary redundancies. Refining database designs is explained in Refining the Database Design.

3. Create a diagram of the database. This diagram should show each table and its relationships with other tables. A database diagram can often give you insight into the structure and dependencies among the major elements of the database you are building. The Database Diagrams subsection explains how to create a database diagram.

4. Determine the best way to store and access the tables in a Unify DataServer/ELS database. To do this, first analyze the types of queries and updates you

want to perform. Then compare what you want to do with your database to the Unify DataServer/ELS access methods, to select the best approach.

Unify DataServer/ELS provides four access methods: hashing, explicit relationships, B-tree indexes, and buffered sequential access. The decision on which method to use is based on how often an item is modified or queried, update speed versus query speed, and the size and complexity of the database.

The Data Access Methods subsection describes the Unify DataServer/ELS access methods in terms of the functions they best support.

7.1 Database elements

The first step in database design is to develop a preliminary list of the elements you need to include in the database. These will become the fields in your database tables.

To develop a preliminary list of fields, talk to the prospective users of the database application. Determine what information users need to maintain and report.

Make a list of definitions and common terms you can use. For example, in companies where some people refer to parts as "items" and some refer to parts as "products," users must agree on the term the database will use to refer to parts.

Do not apply any constraints at this stage of database development. Your goal in this step is to make a list that is as complete as possible. In later steps you can refine the database and limit its scope.

Field data types

As you develop the list of fields for your database design, think about the kind of information each field represents. Is it a name, a date, a price, a quantity, or what? When you build your database, you must tell Unify DataServer/ELS what each field's data type is.

For more information about field characteristics such as internal length versus display length and changing field lengths or types, see Appendix A.

Field data type reference table

The reference table shown in Figure 7.1 summarizes Unify DataServer/ELS database field types, their default display length, and when to use each type. A more complete

description of each field type follows in this subsection. For information about internal data types, see Appendix A.

Field Data Type	Maximum Display Length	Possible Uses, Notes
NUMERIC	9 digits	Ages, quantities, numbers for calculators
FLOAT	20 digits, including decimal places	Real numbers that are not amounts, integer numbers over nine digits
AMOUNT	14 digits, including decimal places	Currency amounts Note: You can specify a length up to 11. Unify DataServer/ELS adds a decimal point and two decimal digits, increasing the maximum length to 14.
DATE	8	Dates from 1 January 1900 to 31 December 2077
LDATE	11	Dates from 1 October 1752 to 31 December 9999
TIME	5	Time of day
STRING	256	Fixed-length alphanumeric data, such as names, telephone numbers, addresses, serial numbers.
TEXT	256	Variable length text descriptions, such as annotated bibliographies or product specifications. Note: You can specify a display length up to 256 characters. The actual length of the field is the length of the ASCII text stored in the field.
BINARY	n/a	Binary data, such as digitized photos and sounds, graphic images, telemetry data, machine instructions, and so on. Note: You cannot specify a display length. The actual length is the length of the binary data stored in the field.
COMB	n/a	Multi-part and primary key fields. This field's length is the total length of its components.

Figure 7.1 Field type reference table

Field data type descriptions

As you develop your list of fields, also think about how you might want to display data. Although each data type has a default display format, you can set environment variables to control display formats for dates and amounts, and to specify editors for text and binary fields.

For more information about setting environment variables, see chapter 5.

Unify DataServer/ELS recognizes several different data types, each of which serves a specific purpose. The following briefly explains the data types recognized by Unify DataServer/ELS, and gives suggestions on when to use each type.

NUMERIC — This is an integer number with a display length up to 9 digits. A null value is stored as 0. Use a NUMERIC data type for a quantity, age, or other set of discrete units that you want to count or maintain statistics on.

If you need to display codes that have leading zeros or embedded dashes, use a STRING data type instead. For example, if you try to enter 0801 in a NUMERIC field, the 0801 converts to 801. If you try to enter something like 541-321, Unify DataServer/ELS won't let you enter the dash. You can enter only a positive or a negative integer (e.g., 541 or -321).

FLOAT — This is a real number with a display length up to 20 digits, including decimal point. A null value is stored as 0.0.

For example, a FLOAT field display length of 170 specifies 17 integer digits to the left of an implied decimal point. A FLOAT field length of 179 specifies nine digits to the right of the decimal point, a decimal point, and seven digits to the left of the decimal point, for a total display length of 17 characters.

Use a FLOAT data type when you need to store a large integer number or need to store to several decimal places, as in a scientific measurement.

AMOUNT — This is a number used to record a monetary amount. An AMOUNT field's display length can be up to II integer digits (Unify DataServer/ELS adds a decimal point and two decimal digits). A null value is stored as .00.

You can specify how AMOUNT fields display by setting the CURR environment variable. This lets you specify the following:

Thousands separator-comma (,), period (.), or blank

Decimal point (.) or comma (,)

Currency symbol position, (left or right of the amount)

Currency symbol (up to three characters, such as $ or DM).

DATE — This is a short date, with a display length of 8 characters. Its default format is MM/DD/YY, but you can set the DATETP environment variable to specify a different order for month, day, and year. A null value is stored as **/**/**.

Use a DATE data type for a date between 1 January, 1900 and 31 December 2077. This data type takes half the physical storage space of the LDATE data type.

LDATE — This is a long date, with a display length of 11 characters. Its default format is MM/DD/YY (DD/MMM/YYYY in PAINT to permit three-letter months). You can set the LDATEFMT environment variable to specify a different order for month, day, and year. You can also specify a different type of month (numeric or alphabetic) and year (2 or 4 characters). A null value is stored as **/**/**.

Use the LDATE data type for dates between October 1, 1752 and December 31, 9999. That is, use LDATE fields for any long-term events, like business plan data forecaster into the next century, or historical records that span centuries.

TIME — This is a time of day, with a display length of five characters. Its display format is HH:MM. Based on a 24-hour clock, HH is the hour from 0-23, and MM is the minutes past the hour from 0-59.

STRING — This is fixed-length alphanumeric data, up to 256 characters in length.

Use STRING data types for names, addresses, short descriptions, telephone numbers, zip codes, serial numbers, or any combination of characters you don't expect to do calculations on.

TEXT — This is variable-length alphanumeric data that can exceed 256 characters in length, such as a word-processed ASCII file. Although

you can only specify a display length up to 256 characters, a TEXT field's internal length is the length of the ASCII data.

The maximum internal length of a TEXT field is limited only by hardware considerations such as disk size. As data is added or erased, the internal length increases or decreases automatically.

The minimum internal length of a TEXT field is 12 characters in the file.db record. The 12 characters specify the location of file.dbv, which contains the actual TEXT field data.

In general, short alphanumeric fields are stored more efficiently as STRING fields, because of the additional disk I/O and processing overhead associated with TEXT fields.

Unlike STRING fields, which are fixed-length and include trailing blanks, TEXT fields contain only the characters you enter. TEXT fields do not contain trailing blanks unless you explicitly enter the blanks. For more information about editing a TEXT type field, see Entering Data in a Text Type Field, in section 15.2.

Use TEXT data types for such things as annotated bibliographies, where you want to store long descriptions that you can index on and search through for specific references.

BINARY

This is binary data. You can store field information in binary files. The display length of a BINARY field is zero (0), but the internal field length is the length of the binary data.

The internal length of a BINARY field is limited only by hardware considerations such as disk size. A BINARY field's internal length grows or shrinks as data is added or deleted. The minimum length of a BINARY field is 64 bytes.

For more information about editing BINARY fields, see Entering Data in a Binary Type Field, in section 15.2.

Use BINARY data types to store data such as bit maps for graphic screen displays, digitized photos, sounds, instrument readings, machine instructions, and so on.

COMB is a special type of database field that is made by combining one or more other fields. Each of its component fields has a data type specified. A COMB field is merely a reference and does not contain or accept input data. Data is accessed by the name of each component field.

You can use a COMB field type to build a single primary key for a table that needs a key made up of two or more smaller fields.

You can also use a COMB field to divide a field into subfields. For example, the field SSN (Social Security Number) can be set up as shown in Figure 7.2:

Field	Key	Ref	Type	Len	Long Name	Comb Field
SSN			COMB		Soc_Sec_Number	
SSN123			NUMERIC	3	Soc_Sec_123	SSN
SSN4S			NUMERIC	2	Soc_Sec_45	SSN
SSN6789			NUMERIC	4	Soc_Sec_6789	SSN

Figure 7.2 Example COMB field

7.2 Refining the database design

Organize your preliminary list of fields into tables. Each table should contain fields that deal with a single subject, such as inventory items or employees.

For each table, determine which field or fields should be used as the primary key, if a primary key is needed. The primary key is the smallest set of fields that uniquely identifies one record in the table.

For example, suppose you have a preliminary design for a warehouse application. The warehouse stocks a number of items for sale, each made by a particular manufacturer. For each manufacturer, you must know its ID number, city, and shipping status. For each item, you must know its manufacturer, serial number, and selling price.

Your preliminary design contains a table with the following fields:

manfitem	
manf_ID	The ID number of the manufacturer
city	The city where the manufacturer is located
status	The relative ease of getting a shipment from a manufacturer
ser_no	The serial number of the item for sale
price	The price charged for the item

Design 1

In Design 1, each item has only one manufacturer, but each manufacturer can make several items, so you might expect manf_ID to be the better choice for a primary key. However, in Design 1 manf_ID cannot be a primary key, because it doesn't uniquely identify one record in the table. The same manf_ID appears once for every item the manufacturer makes.

Eliminating multiple values in fields

The first step in refining the database design is to make sure no table contains fields that can have more than one value. In the first design for the example table, manfitem,

a manufacturer may make up to six items identified by serial numbers, as shown in Figure 7.3

manfitem

Manf_ID	city	status	ser_no	price
0001	Lynn	10	101,102,103,104,105, 106	2.00 . . .
0002	Reston	20	101, 102	. . .
0003	Reston	20	102	. . .
0004	Lynn	10	102, 104, 105	. . .

Figure 7.3 Multiple-valued fields in manfitem design 1

Because ser_no may contain more than one value, this table does not follow the rules of relational database design.

To follow the rules of relational database design, tables should be set up so each field contains a discrete piece of information. Visualize each table as rows (records) and columns (fields), as shown in Figure 7.4. Each row contains one item per column.

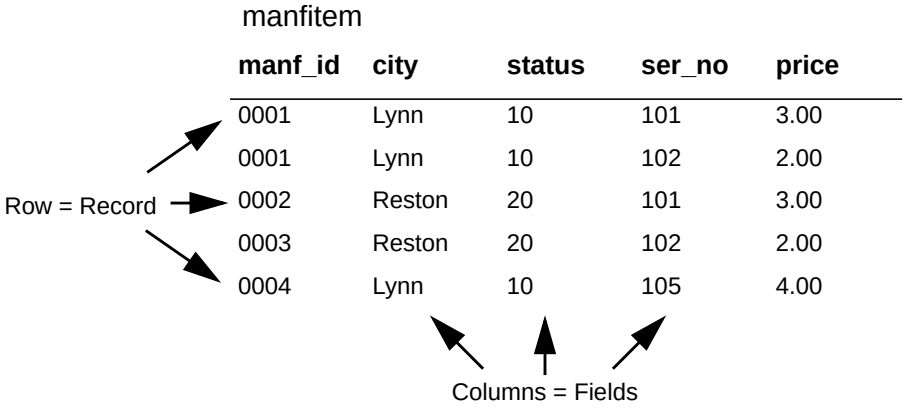


Figure 7.4 Database table visualization

Therefore, to refine the database, the table in Design 1 must be redesigned so the serial number field will never contain more than one value for each record. A possible solution is the table shown in Figure 7.5.

manfitem

Primary Key

manf_ID	city	status	ser_no	price
0001	Lynn	10	101	3.00
0001	Lynn	10	102	2.00
0001	Lynn	10	103	4.00
0001	Lynn	10	104	2.00
0001	Lynn	10	105	1.00
0001	Lynn	10	106	1.00
0002	Reston	20	101	3.00
0002	Reston	20	102	4.00
0003	Reston	20	102	2.00
0004	Lynn	10	102	2.00
0004	Lynn	10	104	3.00
0004	Lynn	10	105	4.00

Figure 7.5 Database Design 2, manfitem table

This refinement, however, means the table now needs both the manufacturer ID and the item serial number to uniquely identify one record. For the record to be unique, the primary key must contain both the manf_ID and ser_no fields. This condition creates a combination primary key.

To determine whether Design 2 is a good relational database design, consider how the design affects the three basic database operations of adding, deleting, and modifying:

1. Adding. Can you add a new record if you don't yet know what items that manufacturer will supply? No, because a combination primary key requires values in each component field. A new manufacturer cannot be added until that manufacturer supplies at least one item. You need both a manufacturer ID and an item serial number to add a record for a new manufacturer.

   For example, you can't add a record for manufacturer 0008 located in Houston, unless you have at least one serial number for an item supplied by manufacturer 0008.

2. Deleting. Can you delete an item that is the only item for a particular manufacturer and leave the manufacturer? No, you cannot delete the item information without also deleting information for the manufacturer's ID, city, and status.

   For example, if you delete item number 102 for manufacturer 0003, you also delete the information that manufacturer 0003 is located in Reston.

3. Modifying. Can you change the location of a manufacturer to a different city? Not easily, because the city for a manufacturer appears in many records. This can cause problems when you are modifying records.

   For example, if manufacturer 0001 moves from Lynn to Revere, you must find every record for manufacturer 0001 and change Lynn to Revere. If you miss even one record, you have manufacturer 0001 located in both Lynn and Revere.

The problem with Design 2 is that too many different kinds of information depend on each other. For example, manufacturer information depends on item information. You should be able to modify manufacturer information without affecting item records, and item information without affecting manufacturer records. These dependencies need to be eliminated.

Eliminating non-key field dependencies

The database in Design 2 obviously needs further refining. Manufacturer information should be separate from item information. To separate the two types of information,

you can divide the manfitem table into two tables, manf and item, as shown in Figure 7.6:

Manf

Primary Key

manf_ID	city	status
0001	Lynn	10
0002	Reston	20
0003	Reston	20
0004	Lynn	10
0005	San Diego	30

item

Primary Key

imanf_ID	ser_no	price
0001	101	3.00
0001	102	2.00
0001	103	4.00
0001	104	2.00
0001	105	1.00
0001	106	1.00
0002	101	3.00
0002	102	4.00
0003	102	2.00
0004	102	2.00
0004	104	3.00
0004	105	4.00

Figure 7.6 Database Design 3, manf and item tables

This revised design overcomes some of the problems involving manf_ID and city in Design 2. The information has not changed, but the design shows some improvements:

1. A new manufacturer can be added even though it does not yet supply any items to the inventory.

   For example, manufacturer 0005 doesn't yet supply an item, but you can add the information that manufacturer 0005 is located in San Diego.

2. The record for an item can be deleted without losing information about the manufacturer.

   For example, the record for manufacturer 0003 and item serial number 102 can be deleted, and the database still has the information that manufacturer 0003 is located in Reston.

3. The location of a manufacturer appears only once in the design. Changing the location of a manufacturer requires only one record update.

   For example, you can change manufacturer 0001's city from Lynn to Revere without having to edit several records.

Yet if you examine Design 3 closely, you should see that it still needs work. Although the item table only contains information about inventory items, the manf table shows a lack of independence among its non-key fields.

For example, the status field depends on the non-key field city, instead of on the key field manf_ID. That is, a manufacturer has a city; and a city has a status. But the status has no real relation to a manufacturer. If the manufacturer's ID number changes, the status is not affected.

However, if a manufacturer's city changes, the status changes as well. This can cause problems.

These dependencies among the manf table's non-key fields cause the following problems for adding, deleting, and modifying data:

1. Adding. You cannot add a record indicating a city's status unless you add a manufacturer for that city. A primary key value is required to add a new record.

   For example, you cannot add the information that Houston has a status of 25 unless you also add a manufacturer located in Houston.

2. Deleting. If the only record for a particular manufacturer is deleted, not only is the manufacturer's information deleted, but also the status information for the city that manufacturer is located in.

   For example, if you delete the record for manufacturer 0005, you lose the information that San Diego has a status of 30.

3. Modifying. The status of a city appears more than once. If you change a city's status, you must change the status for all manufacturers in that city. If you miss a record, you have different statuses for two manufacturers in the same city.

   For example, if you change the status of Lynn from 10 to 20, you must change the status for manufacturers 0001 and 0004.

So the database needs further refinement. To eliminate the problems of status depending on a non-key field, split manf into two tables, manf and cities. The primary key for the new manf table is manf_ID, and the key for cities is city. The new tables have the layout shown in Figure 7.7 .

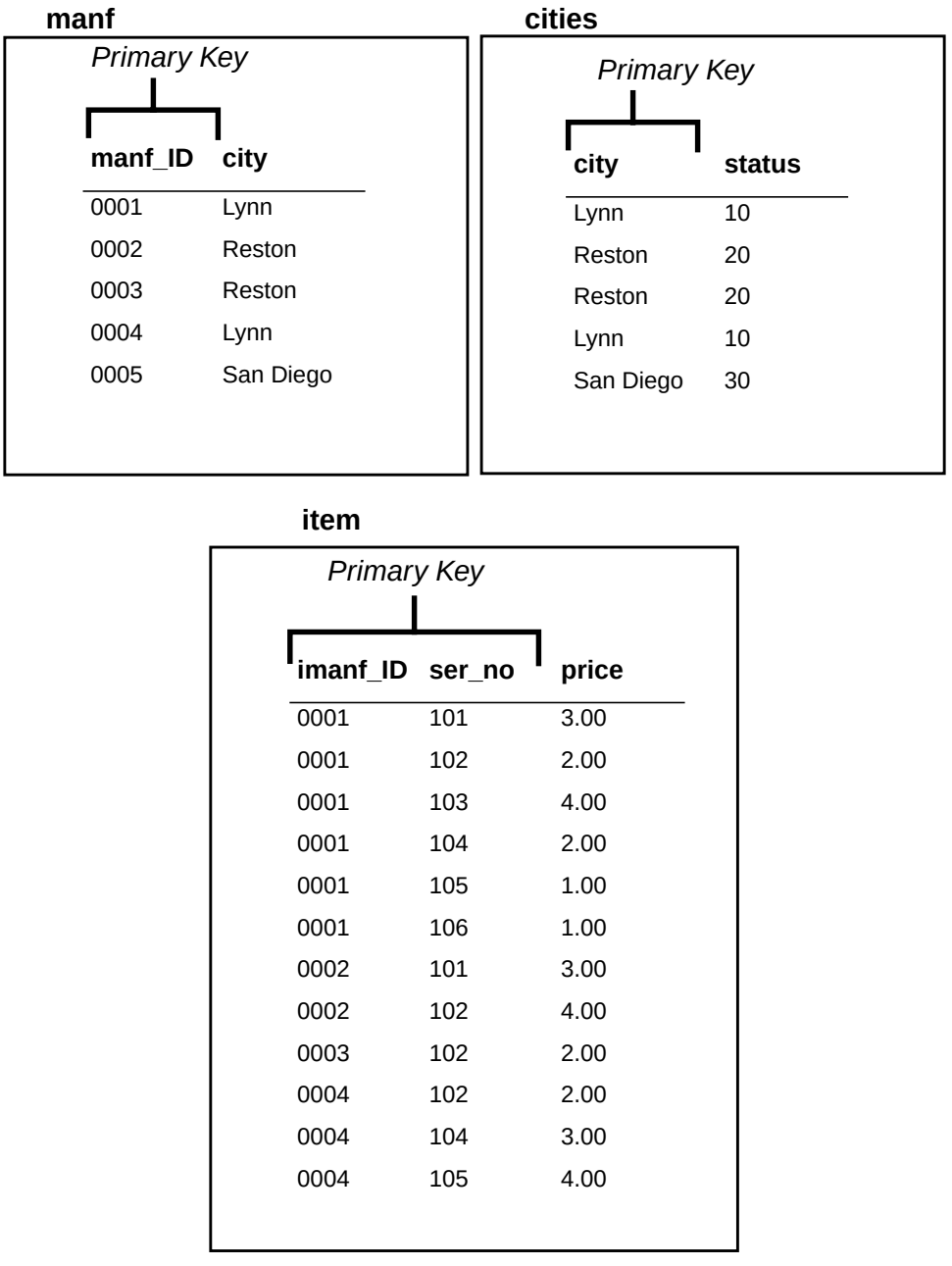


Figure 7.7 Database Design 4, manf, cities, and item table

Now each table in the database design contains information about a single subject. manf contains information about manufacturers; cities, information about city statuses; and item, information about the items for sale.

In Design 4, there is an explicit relationship between mcity in manf and city in cities. The mcity field in manf points to, or references, the key field city in cities. This is shown in Figure 7.8, on the following page, and in Figure 7.9, in section 7.3.

One city can be referenced by several manufacturers. This is a one-to-many relationship. One-to-many relationships are explained in section 7.3, Database Diagrams, and in section 7.4, Data access methods, under Explicit relationships.

Although Design 4 can be expanded, it is a legal Unify DataServer/ELS database design. The design's tables, fields, and relationships are shown in the Unify DataServer/ELS Database design report, Figure 7.8.

DATE: 04/30/99 TIME: 11/29/35 Page 1

DATA DICTIONARY REPORTS
Database Design

TABLE/FIELD	REF	TYPE	LEN	LONG NAME
manf	10			manf
*manf_ID		STRING	4	Manufacturer_ID
mcities	city	STRING	20	City
cities 6				cities
*city		STRING	20	City
status		STRING	2	Status
item 25				item
*item_ID		COMB		Item_ID
imanf_ID	manf_ID	STRING	4	Manufacturer_ID
ser_no		STRING	3	Serial_Number
price		AMOUNT	3	Sales_Price

Figure 7.8 Database design report

For information about database design, see the *Unify DataServer/ELS Direct HLI Programmer's Manual* or any book or article that discusses database design. For a list of suggested sources, see chapter 1, "Introduction to Unify DataServer/ELS".

7.3 Database diagrams

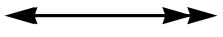
After you have refined your database design, you should draw a diagram showing the tables and the relationships among them. This diagram does not change the definitions of your tables. It merely clarifies the relationships among tables by representing them graphically instead of on several pages of lists.

Relationships can be one-to-one, one-to-many, or many-to-many. For example, in Design 4, one city can be the location of several manufacturers, and one manufacturer can make several items. The tables cities and manf have a one-to-many relationship, and the tables manf and item have a one-to-many relationship.

These relationships can be diagrammed as shown in Figure 7.9:

Figure 7.9 Database diagram, showing explicit relationships

In Figure 7.9, the double-headed arrow indicates "many," and the single-headed arrow indicates "one." Therefore, relationships can be represented using the following symbols:

 indicates a one-to-many relationship

 indicates a many-to-many relationship

 denotes a one-to-one relationship

An example of a many-to-many relationship is a database for scheduling school courses. Each student can have several teachers, and each teacher can have several students.

The purpose of creating a database diagram is to clarify both your and the users' understanding of the database. A diagram is an easy way to represent your database's major elements and their relationships.

7.4 Data access methods

Refining the database design forces you to ask questions about the functional dependencies in an application. From an initial list of table and fields, and with a recognition of which fields are sensitive to or cause changes in other fields, you refine your design until you have a logically consistent database design.

As you refine your design, you ask questions about how the database will be used, what functions it must support, and what reports and types of queries will be run. The answers to these questions determine which Unify DataServer/ELS access methods must be used.

Generally, Unify DataServer/ELS uses the optimum access method for performance, depending on the type of query or operation being performed. You don't actually have to tell Unify DataServer/ELS to use a specific access method each time it searches for information in your database. But you must design your database so Unify DataServer/ELS has available the most efficient access method for the query or operation.

The Unify DataServer/ELS access methods and the operations they are best suited for are as follows:

Hashing
: This method is used to access records randomly by an exact key. The table must have a primary key. If the primary key is a combination key, all parts of the key must be specified exactly.

Explicit Relationships
: This method is used to join tables that reference one another. A relationship exists when a database table references the primary key of another table.

B-trees
: This method is used when a query must find records based on ranges of values or partial, inexact matches. B-trees are defined using a Unify DataServer/ELS utility.

Buffered Sequential Access

This method is used to access every record in the table, starting with the first record and proceeding one-by-one to the last. This method is usually used with raw files.

The following describes each of the Unify DataServer/ELS data access methods and their advantages and disadvantages.

Hashing (primary key)

Unify DataServer/ELS uses a hash index to search by primary key. A table cannot have more than one primary key, and that primary key must be unique.

In Unify DataServer/ELS, a primary key has two purposes in addition to uniquely identifying a record:

- To let you perform a fast random access (hashing)
- To give you something on which to base an explicit relationship.

However, if you don't need to uniquely identify a record, access by hash table, or specify an explicit relationship, a primary key is optional.

If you need fast random access, use as many fields as necessary to get a unique key. But remember that users must enter every component of the key exactly to find a record. For this reason, primary keys should be short, single fields whenever possible.

If users specify an inexact match or range of values, Unify DataServer/ELS uses another access method to find the record.

For Unify DataServer/ELS to find a record in the hash table, it must search for an exact primary key. Unify DataServer/ELS uses a hash algorithm to calculate the relative record number for the record containing the given key value.

The major advantages of hashing are:

1. It is the fastest random access method when the primary key is entered exactly. It is especially fast for short keys of eight bytes or less.

2. Its speed is relatively independent of the number of records in the database table. A hash table access among a million records takes no longer than among 100 records.
3. The overhead in storage space and database updates is moderate compared to that for explicit relationships and B-trees.

The major disadvantage of hashing is:

1. Hashing does not support access to the data in any special order. For example, hashing is not an effective access method for stepping through manufacturers in alphabetical order by name.

Hashing is appropriate for looking up a unique code like an ID number. Specific examples in the Unify DataServer/ELS Tutorial Manual include the manufacturer number, invoice number, and item number.

Explicit relationships

An *explicit relationship* refers to an access method that supports a one-to-many linkage between two tables. If you need to join two tables frequently, an explicit relationship joins the tables faster than any other access method. Consider the linkage between the manf and item tables from Design 4 shown in Figure 7.10:

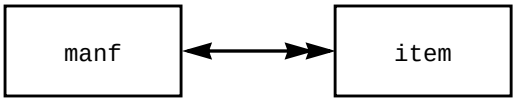


Figure 7.10 Explicit relationship between item and manf

Each manf record has a set of related item records. A linkage exists between two manf and item records if the manufacturer makes the item. In this relationship, the

manf table is the parent, and the item table is the child. Figure 7.11 illustrates this relationship::

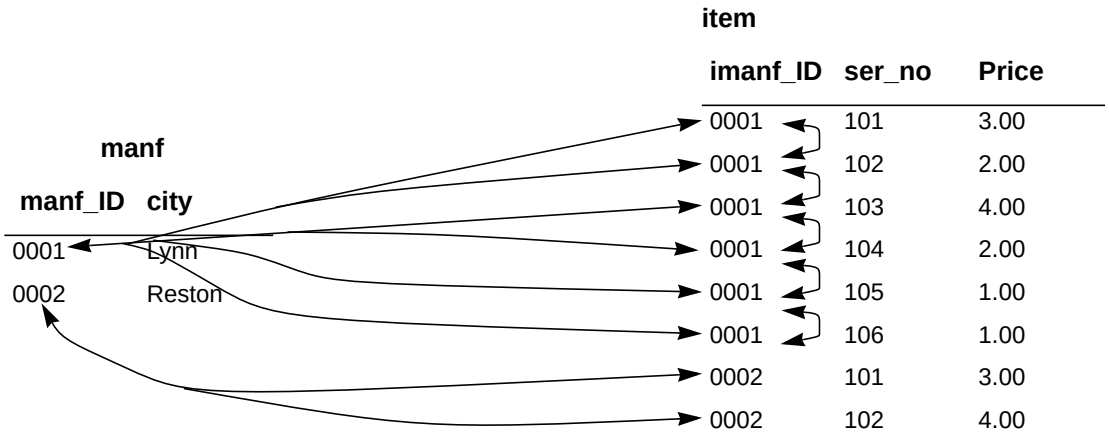


Figure 7.11 Parent-child relationships between manf and item

Besides functioning as a data access method, an explicit relationship also automatically enforces referential integrity. If an explicit relationship exists between two tables, Unify DataServer/ELS only lets you store existing parent key values in the child table. Unify DataServer/ELS does not let you delete a parent record whose key is referenced by child records.

Explicit relationships save storage space and processing time when compared to a B-tree. This is important because a relational database often consists of many small tables. Unify DataServer/ELS must join these tables frequently to process queries. Explicit relationships are the fastest method for this.

In the database design, to declare an explicit relationship between two tables, reference the parent primary key in the REF column of the dependent child field. For example, to declare an explicit relationship between manf and item, place manf_ID, the key for manf, in the REF column for imanf_ID in the item table. Figure 7.12

illustrates this with an excerpt from the Database Design report for Design 4.

TABLE/FIELD	REF	TYPE	LEN	LONG NAME
item 25				item
*item_ID		COMB		item_ID
imanf_ID	manf_ID	STRING	4	Manufacturer_ID
ser_no		STRING	3	Serial_Number
price		AMOUNT	3	Sales_Price

Figure 7.12 Excerpt from report for database design 4

The major advantages of explicit relationships are:

1. Explicit relationships take less disk space and less time to process than do B-trees.
2. Referential integrity is automatically enforced.
3. Given a parent key value, finding and processing each child record is fast and efficient.
4. Given a child record, accessing information about its parent record is fast and efficient.

The major disadvantages of explicit relationships are:

1. Relationship pointer chains are maintained in a LIFO (last-in, first-out) order only, so records cannot be retrieved in a specified sequence without a sort.
2. If you want to add or drop an explicit relationship, you must change the database design and then reconfigure the database.

Properly used, explicit relationships can dramatically improve query performance, at a relatively low cost in terms of disk space and update performance. When joining five or more tables in multi-user applications, explicit relationships can outperform B-trees by a factor of more than ten to one.

B-trees

B-trees have become the standard for indexes in most database management systems for several reasons:

1. B-trees are always balanced, so every search takes the same amount of time.
2. The number of disk accesses required to find a record rises very slowly as the index gets larger.
3. B-trees reorganize themselves automatically, so their performance stays constant even after many additions and deletions.

Data access by B-trees is a modified binary search technique. To understand how B-trees work, it is helpful to review how a binary search operates.

Before a binary search can be used to find records, the records must be sorted. For example, a binary search can be used to find the name "DuPres, Jacques" in a telephone directory. The binary search can find the name only if the list of names has been alphabetized and the name exists. The telephone directory is already sorted, as shown in Figure 7.13.

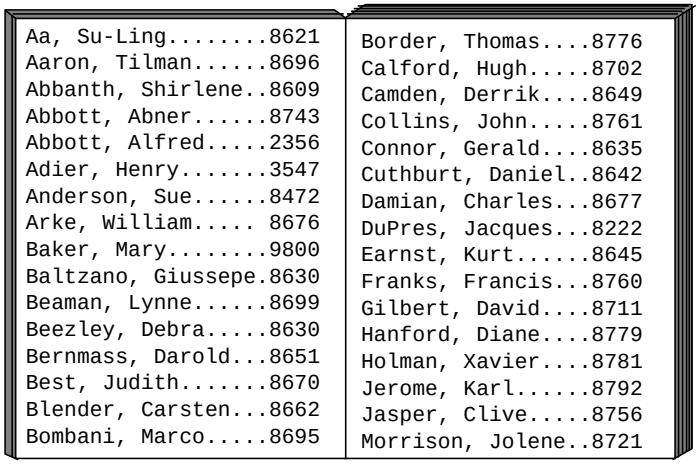


Figure 7.13 Simple telephone directory

A binary search starts by dividing the list of names in the directory into two halves, each containing the same number of names. Suppose that the second half begins with the name "North, David," as shown in Figure 7.14. The first decision in the binary search process is whether "DuPres" occurs *after* "North." The answer is no, it must occur *before*. Therefore, the second half of the directory can be eliminated and the

search restricted to the first half.

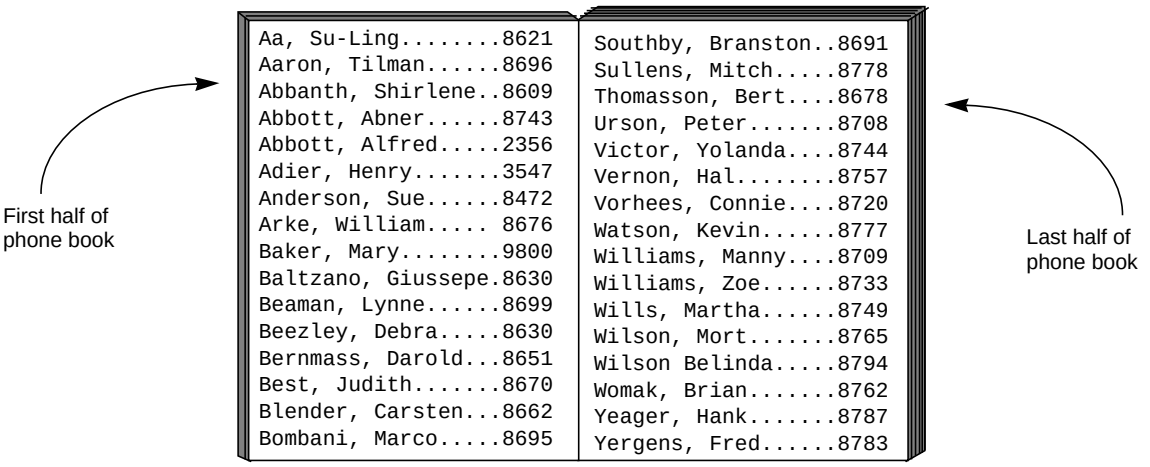


Figure 7.14 Begin binary search of phone book

The divide and search process is repeated for the remaining directory names, as shown in Figure 7.15. The name "Gull, Harold" begins the second half of this new division; *i.e.*, the second quarter of the directory. Because "DuPres" is before "Gull," the second quarter of the directory can be eliminated. At this point, the search space is one-fourth the original size, and only two comparisons have been made.

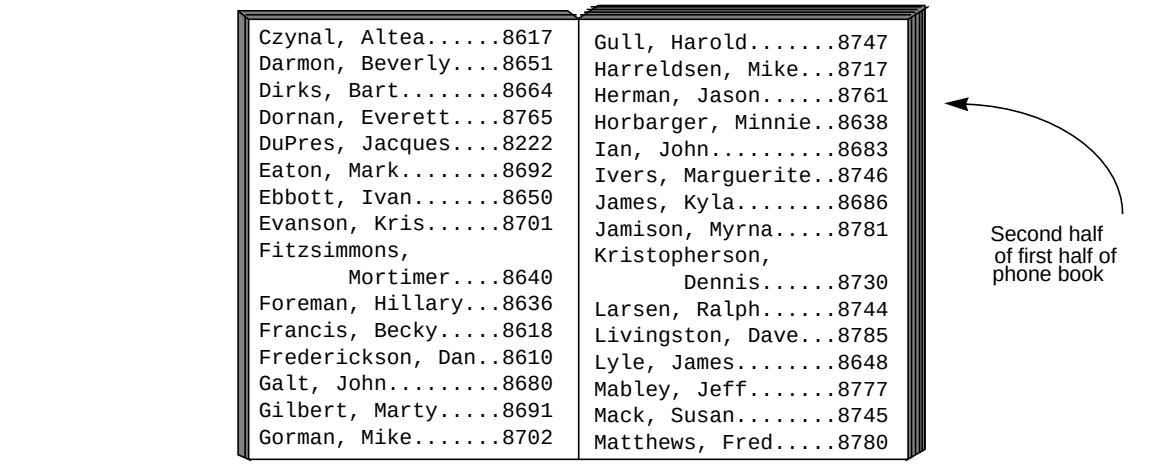


Figure 7.15 Repeat binary search

As the procedure is repeated, the search space is reduced to one-eighth, one-sixteenth, one-thirty-second, and so on, until only "DuPres" is left, shown in Figure 7.16.

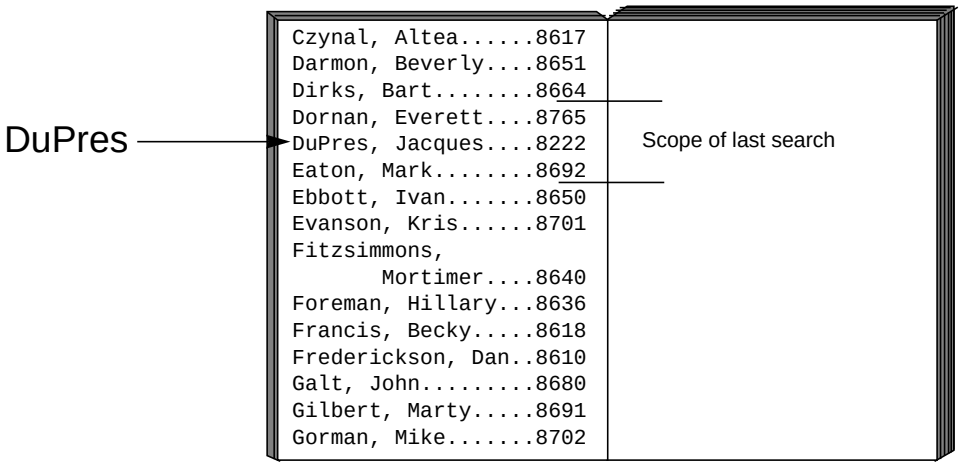


Figure 7.16 Last binary search locating DuPres

A binary search can find one value out of 1024 using no more than 10 decisions. Compared to a sequential search, which requires an average of 512 decisions to locate one value in 1024. A binary search is very efficient.

Even so, some simple modifications can improve the binary search technique. For example, suppose that each decision is a one-of-three comparison. That is, it identifies which third of the search space holds the record you want to find. Figure 7.17 locates the two names that sit on the first and second one-third boundaries in the telephone directory example.

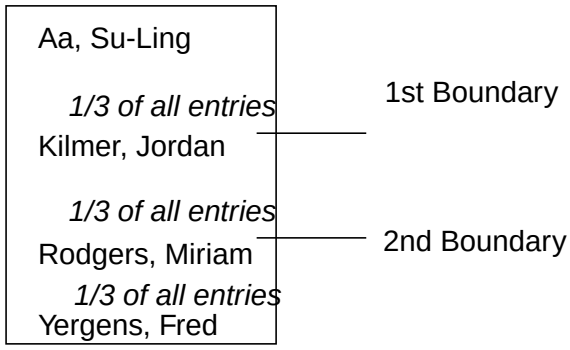


Figure 7.17 A one-of-three decision

The one-of-three decision depends on finding which of the following statements is true:

1. The record is after "Abbott, Alfred" and before "Kilmer, Jordan."
2. The record is after "Kilmer, Jordan" and before "Rodgers, Miriam."

With this one-of-three modification, the records can be located much faster. For example, a search of 1024 records takes no more than seven decisions, instead of the ten decisions required for a binary search.

If a one-of-three search is faster than a binary search, a one-of-four search is faster yet. Similarly, the search gets faster as the N of "one-of-N" gets larger.

The Unify DataServer/ELS B-tree access method uses a one-of-N search, where N is chosen on the basis of the data to be searched, the length of the fields to be indexed, and the disk block size. The search is most efficient when each of the N categories contains an equal number of search objects. B-trees are designed to keep an equal number of entries in each category. They are balanced.

Unify DataServer/ELS lets you build up to 255 B-trees, containing up to eight fields each, within a database. You can create B-trees using the option "Add, Drop B-Tree Indexes."

B-trees excel in three types of queries:

1. Retrieving records in sequence, sorted by the values in the indexed fields.
2. Locating records when the exact value is not known, but the beginning of the value is known. For example, a B-tree can quickly locate all people whose last names begin with "Joh" (as in John, Johnson, and Johnston). However, a B-tree cannot locate people whose last names end with "son" (as in Ellison, Johnson, and Thompson).
3. Locating records in a given range, such as all amounts between $5.00 and $10.00, or all dates between January I and January 31.

B-trees can also be used to speed up joins when you don't want to use an explicit relationship. For example, suppose you have customer and order tables, and the customer number is stored in the order. You can build a non-unique B-tree on the customer number in the order table, and find all orders for a specific customer by searching for the customer number.

The major advantages of B-trees are:

1. B-trees permit ordered access to all records in a given table, based on the values of the indexed fields. When you must access large numbers of records in sorted order very quickly, use B-trees.
2. B-trees speed up queries involving inexact matches, ranges of values, or "begins with" searches. B-trees can be used on any field to create a secondary key, complementing or replacing the primary key.
3. B-trees can be added or dropped without reconfiguring the database.

The major disadvantages of B-trees are:

1. B-trees use more disk space than other Unify DataServer/ELS access methods.
2. Access by B-trees is slower than by hashing or explicit relationships, because B-trees use fairly complex algorithms for searching and balancing.

In most cases, the advantages of B-trees outweigh their disadvantages. However, because of their disk space requirements and processing time, you should establish a clear need before using them. Used correctly, explicit relationships can be used to perform joins much more efficiently.

Buffered sequential access

The most basic access method is sequential access, which scans every record in a table one by one. Unify DataServer/ELS uses two types of sequential access, buffered and nonbuffered.

For nonbuffered sequential access, the operating system reads data from a file one disk block at a time. For buffered sequential access, the operating system reads many blocks of data from a file into an internal buffer, until the buffer is filled.

A file is actually made up of blocks scattered over the disk. The operating system keeps an index of each block's location. When reading from or writing to a particular location in a file, the operating system uses the index to determine which physical block of the disk to use.

Using the index to locate blocks is efficient for files up to one or two megabytes that are accessed randomly and created and deleted often. Such files include text files, program source code, documents, shell scripts or batch files, and executable files.

However, using an index to locate blocks is not efficient for many typical database files. These database files are often larger than ten megabytes, and they often must be accessed sequentially for reports and backups. When a file gets this large, the indexing structure binders performance. To read the correct data block, up to three index blocks must be read first. This can slow down the read process considerably.

The solution for large files is to use the Express I/O file system, often called a raw file. A *raw file* is located in physically adjacent blocks, so the indexing structure is not needed. To create a raw file, unmount one of the file systems, use "Define Database Volumes" (section 9.5) to tell Unify DataServer/ELS about that disk partition, and then perform a "Create Database" (section 8.3).

If the database is made up of raw files, the operating system can transfer raw file blocks directly into a user program's data space. The program can specify a buffer larger than the standard operating system buffer. (The default buffer size is 2048 bytes.) Because raw file disk blocks are physically adjacent, a single disk read can return several blocks of data.

Buffered sequential access can be used for files that are not raw. Disk blocks tend to be scattered in non-raw files, so you must wait between each read for the disk head to find the correct track and then for the correct block to rotate under the head. Using a raw file eliminates these delays.

The major advantages of buffered sequential access are:

1. Buffered sequential access had no disk storage overhead.
2. If used with raw files, a buffered scan is much faster than an unbuffered scan for read-only transaction.

   For more information on raw data file, see the File subsection in chapter 4 and the define Database Volumes subsection in chapter 9.
3. If you must read every record or most records in a table and order is not important, a buffered scan is the fastest method.

The major disadvantages of buffered sequential access are:

1. It looks at every record in the table. If you are only interested in a few of the records, there is no need to read all of them sequentially.
2. The records are returned in a system-defined order. If you need the records in a particular order, you must sort them.
3. If you update most of the records in a table, you negate the advantages of buffering. Each record must be searched for, stored in a buffer, and written to the disk. Nonbuffered sequential access or hashing is more suitable, depending on your needs.
4. Buffered sequential access is designed for reading forward through the file. Although you can read backward using the previous option, this is much less efficient because of how the buffering scheme works.